Compiler Aided Selective Lock Assignment for Improving the Performance of Software Transactional Memory

Abstract *Atomic sections have been recently introduced as a language construct to improve the programmability of concurrent software. They simplify programming by not requiring the explicit specification of locks for shared data. Typically atomic sections are supported by software either through the use of optimistic concurrency by using transactional memory or through the use of pessimistic concurrency using compiler-assigned locks. As a transactional memory system does not take advantage of the specific memory access patterns of an application it often suffers from false conflicts and high validation overheads. On the other hand, the compiler usually ends up assigning coarse grain locks as it relies on whole program points-to analysis which is conservative by nature. This adversely affects performance by limiting concurrency. In order to mitigate the disadvantages associated with STM's lock assignment scheme, we propose a hybrid approach which combines STM's lock assignment with a compiler aided selective lock assignment scheme (referred to as SCLA-STM). SCLA-STM overcomes the inefficiencies associated with a purely compile-time lock assignment approach by (i) using the underlying STM for shared variables where only a conservative analysis is possible by the by the compiler (e.g., in the presence of may-alias points to information) and (ii) being selective about the shared data chosen for the compiler-aided lock assignment. We describe our prototype SCLA-STM scheme implemented in the hp-ux IA-64 C/C++ compiler, using TL2 as our STM implementation. We show that SCLA-STM improves application performance for certain STAMP benchmarks from 1.68% to 37.13%*

1. Introduction

Expressing synchronization using traditional lock based primitives has been found to be both error-prone and restrictive. Locks guarantee isolation only when a program consistently follows a locking discipline. Any violation of the locking discipline leads to concurrency bugs such as race conditions, atomicity violations and deadlocks. Furthermore, lock-based synchronization mechanisms lack composability, which often precludes modular design of concurrent components.

Atomic sections have been proposed recently as a programming idiom for expressing synchronization at a higher level of abstraction than locks. Programmers can specify what code has to execute atomically by simply enclosing the desired block of code with the keyword 'atomic'. Atomic sections are an interesting alternative to locks as they allow local reasoning and are composable.

Atomic sections can be supported in two different ways. One way to do so is to take help from the compiler to transform the atomics to lock based code [1, 2, 3, 4, 5, 6]. This is done through an analysis that determines what locks to associate with what shared data and when to acquire/release locks within the atomic section. Compiler assisted lock allocation (CLA) requires whole program analysis since it needs to know what data is accessed within functions called from inside an atomic section to generate the correct locking discipline.

While a transformation based on CLA can guarantee deadlock freedom and atomicity (under certain assumptions like race freedom), it often ends up allocating the same coarse grained lock to multiple data items (false sharing). This happens because, for indirect data accesses using pointers, CLA schemes need to make a conservative approximation of the data items pointed to by the pointer. This can result in coarse locks if the alias sets contain a large number of may-aliases. Also, CLA needs to make conservative approximations when shared data is passed to opaque external library functions accessed inside atomic sections. Last, and most importantly, all lock based implementations are pessimistic in nature, and require the locks to be acquired before an atomic section is executed, and the cost is incurred irrespective of whether or not there is a conflicting atomic section.

The alternate way of supporting atomic sections is to rely on an underlying software transactional memory implementation [8, 9, 10, 11, 12, 13, 23]. STMs allow for optimistic execution by allowing multiple atomic sections to run concurrently assuming they will not conflict. However, in case a conflict does occur they have a mechanism to detect and recover from such conflicts [10]. Below we briefly describe how STMs operate.

To enable conflict detection STMs track metadata for each data item accessed within an atomic sections at runtime. To avoid having to collect huge amounts of metadata they combine data items together either by mapping the data words being accessed to a hash table or by treating data belonging to the same object as a single entity.

To prevent races among metadata updates, STM implementations typically employ fine grained locking to lock at a per metadata level (locks could be acquired explicitly or by using low level atomic operations like CAS or LL-SC). The mapping from data items to metadata can therefore be thought of as a mapping from data items to locks. Typically STMs use a hash function to map the address of the shared data item to a lock [11, 12].

The actual process of detecting conflicts (often referred to as read/write validation) typically involves going through all the accumulated metadata to see if there has been a read-write or write-write conflict. The cost of conflict detection therefore depends on the efficiency of the mapping from data items to locks. Once a conflict is detected the STM then chooses to abort one of the atomic sections and rolls back the effects of it.

Lock assignment (LA) purely by the software transactional memory implementations typically done at run-time do not have any knowledge of the application's data access patterns. Since the number of locks available for assignment is often limited, multiple uncorrelated data items can get mapped to the same lock. This can result in 'false conflicts'. Such false conflicts will result in increased number of aborts/rollbacks and can impact execution time [14, 15].

The use of fine grained locking can also lead to high read validation costs and lock acquire costs [17, 26]. This can impact execution time adversely, for transactions which touch a large volume of shared data. On the positive side, STM's runtime lock assignment (RLA) scheme does not require whole program analysis nor is it dependent on compiler's alias analysis capabilities unlike any CLA scheme.

The pitfalls of application unaware RLA employed by STMs leading to false conflicts has been studied in [14, 15]. Yoo et al [16] propose an improved hash function to address false conflicts. Their scheme reduces the space required for storing the ownership records without increasing the false sharing by packing more transaction records into each hash table entry. However their modified hash function does not take into account the data access patterns of the shared data in the application's atomic sections and

enforces mutual exclusion at cache line granularity level uniformly.

The disadvantages associated with CLA and STM's RLA schemes originate from the fact that all the shared data items of the given application are assigned locks using either a purely compile time analysis or a purely runtime STM driven hash-function approach. Such a "One Approach fits all" principle does not match with the natural layout of shared data structures in all applications. Some of the shared data items are typically more amenable to CLA, which can take advantage of the compiler's knowledge of the application's data access patterns, while some others are more amenable to STM's fine grained RLA scheme.

In this paper we propose a hybrid lock assignment scheme to address this issue. In our approach, the compiler uses inter-procedural whole program static analysis to assign locks to selected shared data items (those for which it can reason accurately and compile time assignment would be beneficial) while other shared data items are covered by the default STM lock assignment scheme. Our approach will be referred to as the Selective Compiler assisted Lock Assignment based STM (SCLA-STM). In our hybrid scheme, we do not add any instrumentation to acquire/release the required locks. Only the lock mapping is generated at compile time and is communicated to the underlying STM which performs the lock acquires as per its optimistic concurrency algorithm.

To ensure that the hybrid scheme is safe, a clean handshake between the compiler and the STM interface is essential. We describe the extensions needed in the STM to facilitate this and show that our approach preserves the original semantics of the underlying STM implementation. We have implemented a prototype of our scheme in HP-UX IA-64 C/C++ compiler using TL2 [11] as our underlying STM implementation for our experimental evaluation. Results indicate that our approach performs better than the default STM's RLA scheme, which is application unaware. We show that acting in compliment to the STM's default RLA scheme, our SCLA-STM scheme can improve application performance in 4 of the STAMP [20] benchmarks from 1.68% to 37.13% over the base STM implementation, while reducing the percentage of aborts from 1.43% to 29.9%.

This paper is organized as follows: In Section 2, we provide the necessary motivation for our SCLA-STM scheme. Section 3 describes our scheme and discusses the issues in a practical implementation. We report the results of our experimental evaluation in Section 4. We discuss related work in Section 5 and conclude with a short summary in Section 6.

2. Motivation

This section motivates the need for a hybrid scheme by illustrating the drawbacks of using a pure CLA or RLA approach.

2.1 Issues with Complier Assisted Lock Allocation (CLA)

CLA schemes are dependent on the underlying alias analysis [18, 19] of the compiler. For instance, consider the following code sample in Fig 1. The example in Fig 1 simply increments one of 2 counters depending on the value of b. A CLA scheme employing an alias analysis which does not disambiguate between individual array elements will result in allocating same lock L to g[0] and g[1]. This can result in false conflicts between callers invoking increment(true) vs. increment(false). The reason is that L covers both the variables g[0] and g[1] whereas it is correct to cover g[0] and g[1] with separate locks. The false conflict is induced by the imprecision of the underlying alias analysis of the CLA scheme. The compiler reports that the pointer may alias with both g[0] and g[1] and therefore conservatively assigns both of them the same lock. On the other hand, a lock assignment by an STM implementation which operates at word granularity will assign separate locks for g[0] and g[1] as these are 2 different data addresses and hence they get mapped to 2 different locks. This will avoid the false conflicts between the caller of increment(true) and the caller of increment(false).

```
int g[] = {0,0};

void increment (Boolean b) atomic
{
        if (b) g[0] ++;

        else g[1]++;
}
```

Figure 1. Example 1

Even if the CLA scheme has support for fine grained locking, the very nature of static analysis based lock assignment can result in excessive locking. Consider the above code snippet in Fig 2.

```
1.  struct node { struct node* next; int* dataptr;};
    typedef struct elem elem;
2.  elem* x,y, m;
3.  int* w;
4.  elem* p;

5.  p = m->dataptr;
6.  if (....) x = y;

7.  atomic {
8.          x->dataptr = w;
9.          int* z = y->dataptr;
10.         *z = null;
11.         *p = 10;
12. }
```

Figure 2. Example 2

In the example code, compiler can do a backward analysis to determine that z is equivalent to y->dataptr. The expression y->dataptr can be affected by the update of x->dataptr in line 8 if x and y are aliased. Infact x and y will be aliased if the branch in line 6 is taken. 'z' at line 10 can point to y->dataptr or w based on the branch outcome. Since the compiler cannot determine whether this branch will be taken or not at compile time, it needs to be conservative. Hence a CLA scheme will assign the same lock to both w and y->dataptr. On the other hand, STM's runtime locking scheme will need to acquire only one lock out of the two locks which are mapped to y->dataptr and w. Thus CLA suffers due to the may-alias of x with y. On the other hand, consider the access *p in line 11. p is must-aliased to m->dataptr. Hence a CLA scheme can correctly infer that the lock covering the data item m->dataptr needs to be acquired at line 11 before dereferencing p. Thus must-aliases pose no issues for CLA schemes unlike may-aliases.

2.2 Issues with Runtime Lock Assignment (RLA) by STM

Lock assignment by STM takes no note of the access pattern of the data inside the atomic sections when doing lock assignment. Also, since the locking granularity in STM is extremely fine

grained, the conflict detection costs can be high for a transaction with a large memory footprint.

Let us consider the following code snippet in Figure 3 Example 3, which is simplified from the STAMP benchmark program 'kmeans'. In the code snippet, each dynamic instance of the atomic section accesses an independent array section. Consider an STM implementation which uses word based granularity locking with a lock array of size 100. Each atomic section is independent and operates on one independent chunk of the array. Since the shared data array size is 1000, the STM needs to map 1000 different data elements to the limited 100 locks. Depending on the hash function used, different and unrelated data map to same lock resulting in false conflicts. Also each dynamic instance of atomic section will end up performing 100 lock acquires. On the other hand a CLA scheme, which can determine that each dynamic instance of the atomic section accesses an independent array section and the start and end of the array section operated in each atomic section, will assign a lock per each independent array section.

```
#define PERTHREAD_CHUNK 100

int array[1000];
int num_threads;

foo()  {
   for (int thread_id = 0; thread_id < 10; thread_id++)  {
      int chunk_start  = thread_id * PERTHREAD_CHUNK;

      pthread_create(bar, &array[chunk_start], 100);
   }
}

bar (int* array_section, int my_chunk) atomic {
   for (int i = 0; i < my_chunk; i++) {
         *(array_section + i) = ….
   }
}
```

Figure 3. Example 3

This results in zero false conflicts since each array section gets a unique lock. Each atomic section requires only one lock acquire, resulting in lowered lock acquire overheads and read validation costs. Thus, a CLA scheme driven by compiler analysis of application's data access pattern can assign a better lock mapping in this case compared to the default STM's RLA.

Note that false conflicts can also happen in object granularity STMs. For example consider the following 2 atomic sections in Figure 4.

```
Atomic                          Atomic
{                               {
   int temp=node->left;            int temp=node->right;
   node->left=global2;             node->right=global1;
   global1=temp;                   global1=temp;
}                               }
```

Figure 4. Example 4

If the STM were to use object granularity locking it would prohibit these 2 atomics from executing in parallel. However if a compiler maps the data item to separate locks one can gain concurrency.

2.3 A Hybrid Approach

It is easy to see that the various example code snippets illustrated above can be part of a single atomic section in real life applications. Hence some of the shared data in an atomic section exhibit access patterns which are better exploited by CLA whereas certain other shared data are more amenable to STM's fine grained locking assignment. Hence we propose a hybrid lock assignment scheme over an underlying STM implementation, wherein compiler assigns locks selectively to certain shared data items whereas other shared data items are covered by the default STM lock assignment.

Considering the example in Fig. 3 from Section 2.2, we can see that STM needs to perform 100 read validations and 100 number of lock acquires for each dynamic instance of atomic section. Given the size of shared data array to be 1000 and the size of the STM lock array to be 100, we can see that multiple array elements will map to the same lock by the hashing function due to the limited size of the lock table array. Since STM's lock assignment scheme has no knowledge of the access pattern of the shared data in the atomic section, it has no way of figuring out that all array elements in a given array section are accessed together in the atomic section and that each atomic section accesses independent elements. Hence in this case it would be sufficient if the STM assigns only one lock for each section of the array, but this information is not available/inferable by the STM.

However the compiler has full knowledge of the application and can detect the access pattern of the shared data in the atomic section. Compiler analysis can recognize that it is sufficient to assign a single lock for each distinct array section accessed in this critical section thereby reducing the read validation and lock acquire costs and reducing false conflicts. Our approach essentially captures this information and gives it to the STM, so that for each of the array chunks accessed by the different threads a single lock is mapped to it. This reduces the overheads of having to acquire 1000 locks to 10 locks.

Now consider the example in Fig 2 from Section 2.1. Since our hybrid approach assigns compile time locks selectively, it can decide not to assign compile time locks for this and allow the access to *z in line 10 of Fig 2 to be handled by the STM, thereby avoiding the issue encountered in pure CLA schemes due to may-aliases.

Our hybrid approach selectively perform compile-time assignment of locks for some the shared variables where such an approach would be beneficial, and relies on runtime lock assignment and the underlying the STM model to handle the other shared variables appropriately.

3. Selective Compile time Lock Assignment Scheme

In this section, we describe our selective compile time lock assignment scheme for STM in detail.

3.1 Overview of our Approach

Our SCLA-STM scheme consists of the following steps:

a) Selection of shared data items for lock assignment by the compiler. Note that a purely compile time lock assignment scheme needs to assign lock mapping for all shared data. Whereas our approach, can selectively do the lock assignment so as to avoid false conflicts that traditional CLA

is susceptible to. We describe our criteria for selecting data in Section 3.2.

b) Assignment of locks by the compiler to the selected data.

c) Creation of the required locks by the compiler. Locks are created either statically (for statically allocated data) or by inserting code to allocate the locks dynamically for dynamically allocated data.

d) Communicating the compile time lock assignment to the STM. This is achieved by the handshake mechanism set up between the compiler and the STM for communicating the lock assignment done by the compiler, details of which are described in Section 3.5.

Note that a pure CLA scheme [1, 2, 3, 4] needs to adds instrumentation at compile time to acquire the required lock before accessing the shared variable (typically this is done at the entry to the atomic section), whereas in our hybrid scheme, only the lock mapping is generated at compile time and code is added to communicate this mapping to STM. We do not add any instrumentation to acquire/release the required locks. Only the mapping is communicated to the STM which performs the lock acquires as per its optimistic concurrency algorithm.

3.2 Candidate Selection for SCLA-STM scheme

Compiler performs an inter-procedural analysis of the application and creates a list of shared data accesses made in each atomic section. During inter-procedural analysis, compiler performs points-to analysis [18, 19] on the program and maps each pointer in the program to a points-to set. The points-to set consists of the set of locations that a pointer access points to. Two types of alias relationships are possible, namely must-alias and may alias. Must alias relationship is an alias relationship that holds true for all executions of the program, whereas a may-alias relationship is one that holds true for some execution of the program P. Looking back at our example of Fig 2, p is must aliased to m->dataptr whereas x is may-aliased to y. We select only those accesses whose points to set consists of only must-aliases for compiler lock assignment. The intuition behind this choice is that these are the set of locations for which compiler has definitive alias information which holds good for all execution paths and hence the compile time lock assignment will be effective. In order to ensure that every shared location is protected by a lock consistently throughout the entire execution of the application, our SCLA scheme does the following:

i. First, for each of the shared accesses ‘p’ which has only must alias entries in its points to set, shared data corresponding to each of these must-alias entries will be selected for CLA and will be assigned a compile time lock. We refer to such shared data items selected for SCLA as SCLA data items. For SCLA data items, STM uses the compiler assigned lock mapping.

ii. If a shared access ‘q’ has a may-alias to a SCLA data item in its points-to set, then compiler instruments such memory references, so that at runtime, when the lock is required for the shared access ‘q’, first STM checks if a CLA lock mapping is available for this data address. A compiler assigned lock mapping will be available if ‘q’ in fact points to a SCLA data item at runtime. In that case, STM uses the CLA assigned lock mapping. Else the default STM lock assignment as computed in an unmodified STM will be used.

iii. For a shared access ‘r’ whose points-to set does not include any SCLA data item, the default STM lock assignment as computed in an unmodified STM will be used .

We illustrate this with the following example: Consider 3 shared accesses ‘p’, ‘q’ and ‘r’ such that

a) ‘p’ whose points to set has {must alias (global1)};

b) ‘q’ whose points to set consists of {may alias (global1), may_alias(global3)};

c) ‘r’ whose points to set has {may_alias(global4)}.

As per step i above, global1 is selected for SCLA. By step ii above, q is instrumented by the compiler so that at runtime STM first checks to see if a compiler assigned lock is available for that address. If q in fact points to global1 at runtime, then a compiler assigned lock mapping will be found and hence it will be used. If q points to global 3, then it will not have a compiler lock mapping. So STM will use its default STM lock assignment. For the shared access ‘r’, STM will use its default STM lock mapping as per step iii above.

3.3 Lock Assignment for Selected Candidates

The inputs to the problem are

1. A set D of selected candidates (d_1, d_2… d_n)

2. A set AS of atomic sections (AS_1, AS_2…AS_m)

3. A mapping SD from an atomic section to candidate data accessed within it. The candidates accessed within the i^{th} atomic section is represented as an array SD_i.

The problem now is to create a set of locks L= (l_1, l_2…l_n) and a locking discipline LD that maps each selected data di to a lock LD(di). As a corollary, each atomic section ASi gets assigned a set of locks ASLock (AS_i) which is a union of the individual locks of the data accessed in AS_i. While a lock assignment which assigns a separate lock to each candidate will be sound, this naïve solution can result in excessive validation overhead. Hence we would like to use the fewest set of locks without sacrificing parallelism. Therefore the problem we want to solve is “Given the sets D, AS and SD, construct a minimum set of locks L and a mapping LD that maps each shared data item to a lock, so that no two atomic sections need a common lock unless they access common data.”

Mathematically minimize |L| such that

i. for all d_i, $LD(d_i) = l_j$ where $l_j \in L$

ii. for all i and j, $ASLock(AS_i) \cap ASLock(AS_j) = NULL$ if and only if $SD(i) \cap SD(j) = NULL$

We approach the problem in a manner similar to the register allocation problem [21] by constructing an interference graph that captures the relationship between shared data. Each shared datum is assigned a separate node. All must aliases of an access are assigned to the same node. We then add edges between two nodes if assigning them the same lock may inhibit concurrency. Specifically, an edge is added between nodes A and B if and only if both the following constraints are met

I. there exists an atomic section that accesses A but not B, and

II. there exists an atomic section that accesses B but not A.

Any valid coloring of the graph is a solution for the problem of finding a locking discipline that does not of hamper concurrency. Smaller the number of colors used the lesser will be the validation cost. To color the graph we use the heuristic procedure used by Chaitin et al [21] for register allocation.

3.3.1 Constructing the Interference Graph

The simplest way of constructing the graph will involve going through each pair of nodes and figuring out if they conflict or not. To determine conflicts we have to go through each atomic section until two atomic sections that respectively satisfy constraints I and II are found.

For each pair of (di, dj) of shared data items,

i. initialize constraint1 = false; constraint2 = false;
ii. Walk through each atomic section A € AS, If di is accessed in A, but not dj, then set constraint1 = true; If dj is accessed in A but not di, then set constraint2 = true;
iii. If both constraint1 && constraint2 are set to true, add an edge between di and dj

This simple algorithm has a complexity $O(|AS| . n^2)$ where n is the number of shared data items, as each pair of shared data needs to be checked, in the worst case, in every atomic section

3.3.2 Example

The algorithm is best explained with an example. Consider a sample program which contains 5 atomic sections with accesses in each atomic section being {d2, d3, d4}, {d2, d5}, {d2, d3, d4, d5, d6}, {d3, d4} and {d1, d6} respectively. We have the interference graph constructed as shown in Fig 5 which can be colored using 3 colors (the colors are shown in brackets in each node) as shown in Fig 5 below.

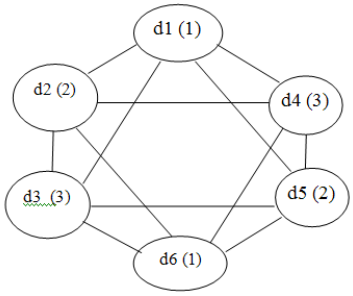

Figure 5. Interference Graph

3.4 Lock Allocation

At the end of lock assignment phase, the compiler has a lock mapping which determines which candidate data are covered by the same lock. Now the compiler needs to allocate the locks to satisfy this assignment and communicate the mapping between the shared data address and the address of the compiler created lock to the STM. For statically allocated candidate data, the compiler can create the locks statically and create the mapping between the datum address and lock address. While the locks are created statically, the mapping is established at the start of the program. For dynamically allocated data, compiler inserts code to dynamically allocate the locks and create the mapping between the dynamically allocated data and the lock, immediately after the allocation of the dynamic data.

There could be another class of scenarios where the shared locations are not determined in advance. In that case, the compiler inserts instrumentation at the point the shared locations are available. Though this class is not very common, the overhead of instantiating the mapping at runtime could be noticeable. If that's the case, the compiler could take into account the instrumentation overheads while heuristically choosing the locations to apply static lock allocation to.

Compiler also generates code to free the dynamically allocated locks at program exit. While it is possible to free the dynamically allocated locks at the point where the corresponding data items are freed, the issue of lock recycling needs to be considered and handled in a manner similar to the handling of dynamically allocated shared data whose memory can be recycled. For our current implementation, we work around this issue, by freeing the dynamically allocated locks only at program exit, so that compiler assigned dynamically allocated locks are not recycled. Though the increase in overall memory consumption due to this workaround was not significant for the benchmarks we studied, this factor might need to be taken in to account while building a production quality SCLA.

3.5 Communication of Lock Assignment to the STM

We use TL2 [11] as our underlying STM implementation and hence our discussion below is more closely coupled to TL2 implementation. TL2 is a library based STM implementation as opposed to a compiler based STM implementation. The ideas in this section are applicable to both library based and compiler based STM implementations though the actual implementation specifics may differ. We first describe briefly the default lock assignment scheme in TL2.

TL2 is a word based STM with invisible reads with global version number based validation. It uses versioned write locks to protect shared memory locations. It has a global version clock variable which will be read and incremented by each writing transaction and will be read by each read-only transaction. In its simplest form, the versioned write-lock is a single word spin-lock that uses a CAS operation to acquire the lock and a store to release it. Since only one bit of the lock word is needed to indicate that the lock is taken, the rest of the lock word is used to hold a version number. An array of lock words is allocated by TL2. The size of the lock table array is 2^20 entries. To obtain the lock word associated with a given shared memory location, the STM uses a hash function of the memory address. The mapping of a memory address of the shared datum 'Addr' is given by

(Base of the lock table array + ((((Addr)+128) >> 2) & (LOCK TABLESIZE -1)))

Note that the assignment of locks to the shared datum is purely dependent on the hash function.

Since the compiler created lock assignment needs to be communicated to the STM, the STM implementation needs to be augmented with a new data structure known as Compiler Lock Assignment Table (CLAT). STM's Application Programming Interfaces are extended with two new functions, namely *GetCompilerLockMapping* and *SetCompilerLockMapping*. *SetCompilerLockMapping* can be invoked for inserting a mapping into the CLAT and *GetCompilerLockMapping* can be used to return the lock mapping for a given datum from CLAT. Compiler inserts calls to *SetCompilerLockMapping* for recording the lock mapping for the data items selected by SCLA-STM scheme.

Typically STM implementation supports an internal *GetLock* Function which given a datum address, returns the lock address corresponding to it. For TL2, *GetLock* function is implemented by means of a macro known as *PSLOCK*. Given a datum address, *PSLOCK* returns the address of lock entry using the STM's hash function. The transactional load and transactional store interfaces (TxLoad and TxStore in case of TL2) invoke the PSLOCK function to obtain the lock address.

In our SCLA-STM scheme, STM is augmented with two new interfaces '*TxLoadWithCompilerLock*' and '*TxStorewithCompilerLock*'. These interfaces query CLAT first and if there is no mapping available in CLAT for that address, they will return the STM"s default lock mapping using the *PSLOCK* function. Except for this one change, the functionality of the interfaces '*Tx*WithCompilerLock*' is exactly identical to that of Tx*.

For those shared accesses whose points to sets include a datum selected for CLA by our hybrid approach, compiler replaces the calls to TxLoad/TxStore in atomic sections with calls to '*Tx*WithCompilerLock'*, so that CLAT is consulted first to obtain the lock for these data by the STM. If no mapping is available in CLAT for that address, STM defaults to its runtime lock assignment for that access. Next we explain why this approach is sound.

3.6 Preserving the Semantics of Atomic Sections

Below we argue that our approach preserves the "all or nothing" semantics of atomic sections. Our hybrid lock assignment scheme is built on top of an existing word based STM (TL2). As TL2 preserves the atomic semantics all we need to show is that we do not introduce any transformations that violate the guarantees provided by the underlying STM. The only change our scheme incorporates is in the way addresses are mapped to locks. On this front, the underlying STM requires the guarantees that:

(i) There is a locking discipline that maps each shared address accessed within atomic sections to a lock.
(ii) The locking discipline is consistent. Assuming that any of the atomic sections can be executing concurrently this implies that each access to a shared address should be consistently mapped to the same lock.

Both these conditions are guaranteed by our construction.

The accesses to shared locations accessed within atomic sections can be partitioned into those accessed via must aliases and those accessed via may aliases. Consider the must alias case: since the compiler has precise information, the lock will be found in the CLAT at runtime. Since the mapping from address to lock is always the same, the above two criterion are satisfied for the references that have must-alias. For the may-alias case, at runtime the address will either resolve to a location corresponding to the CLAT or PSLOCK. In either case the address will definitely map to a lock, so condition (i) is satisfied. Further, note that the CLAT is first queried in this case. So every time the address resolves to a location for which the compiler determines the mapping, the correct lock will be found. Otherwise, the lock will be obtained from PSLOCK which always assigns the same lock for a given address throughout the entire execution of the program. Hence the locking discipline is consistent and (ii) is satisfied.

3.7 Overheads of Our Approach

SCLA has following sources of overheads on the application's execution time compared to an unmodified STM implementation.

a) Extra code that needs to be executed for allocating compiler assigned locks, recording the mapping with the STM and for looking up the mapping from CLAT. Recording the mapping into CLAT is a one time overhead.
b) Additional dynamic memory requirements for dynamically allocated compiler locks.
c) Cache overheads due to CLAT accesses

SCLA can be augmented with a heuristic, which estimates statically the overheads associated and adds a threshold to limit the number of candidates selected. We plan to investigate this as part of our future work. We report the overheads observed in Section 4.3.

3.8 Compiler Support for In-Place Locks

If the lock assignment scheme protects all fields of a structure with the same lock (similar to object level locking applied to structures), then it is possible to assign the lock along with the object as an in-place lock. In-place locks have the advantage of spatial locality and hence reduced d-cache misses compared to external locks. We used the whole program structure layout (WPSL) optimization phase of the optimizer [25] to perform this optimization. If there are instances where the structure is passed to opaque functions or address arithmetic is performed on the members of the structure, it would not be possible for the compiler to perform this optimization. These legality checks are the same as those used in the whole program structure layout framework [25]. In case the legality checks are not satisfied for the addition of the lock field to the structure, compiler's WPSL framework does not perform this optimization. Instead the compiler allocates locks external to the structure. In section 4.2, we report performance improvements obtained by using in-place locks.

3.9 Our Compiler Framework

We have implemented the prototype of our SCLA-STM scheme in the HP-UX IA-64 C/C++ compilers. SYZYGY IA-64 C/C++ compiler [27] has a whole program inter-procedural analysis (IPA) phase with support for whole program points to analysis. We implemented the SCLA prototype in the IPA phase of the compiler. Note that we used TL2 a library based STM, as our underlying STM implementation. Hence all the applications which can run on TL2, will have the transactional loads/stores marked explicitly (this is the case with the STAMP benchmarks which we used for our experiments). Therefore the compiler need not perform any explicit step to infer which of the accesses inside an atomic section are to shared data. In a compiler based STM implementation, this is not the case. In case of compiler based STMs, before performing SCLA, shared data inside each atomic section needs to be identified explicitly by compiler analysis [13].

4. Experimental Evaluation

4.1 Experimental Methodology

To evaluate the effectiveness of our hybrid scheme, we used the STAMP benchmark suite [20] version 0.9.10. STAMP comprises of 8 applications: kmeans (an implementation of K-means clustering), genome (a gene sequencing program), bayes (a Bayesian network learning program), labyrinth (a maze routing program), vacation (a client/server travel reservation system), intruder (signature based network intrusion detection), ssca2 (four kernels that operate large weighted directed multi-graph) and yada (yet another delaunay application). We implemented the SCLA-STM prototype in the HP-UX IA-64 C/C++ compiler. For the

Benchmark	Number of static instances of atomic sections	No. of dynamic instances of atomic sections	% of aborts for total transactions in unmodified STM (with 8 threads)	Data structures accessed in atomic sections	Atomic sections contributing most % of the total aborts
kmeans	3	11747913	43.8%	2 integer arrays, 2 globals	Normal.c:168 – 96%
vacation	3	4256610	38.65%	4 RB trees	Aborts distributed over all atomic sections
genome	5	2503393	1.06%	Linked list	Sequencer.c:394- 99%
intruder	3	62484204	55.75%	Queue, linked list	Intruder.c:226 –61%
labyrinth	3	1219	16.1%	3d array	Router.c:396 – 99%
Ssca2	10	22362791	0.3%	Array	computeGraph.c:435
yada	6	53745	46.1%	Structure element	Region.c:333 – 54%
bayes	15	2247	4.1%	Linked list, structure learner	Learner.c:1202

Table – 1 Characteristics of STAMP Benchmark Programs

performance runs, the benchmarks, compiled with the SCLA-STM scheme enabled, were run on the modified TL2 (as described in section 3) implementation. The baseline for our experiments is to compile the benchmarks at the same optimization level (level 4) with inter-procedural analysis enabled, but with SCLA-STM phase turned off in the compiler and runthem on the unmodified TL2 implementation. We used the native (non-simulator) input sets of the STAMP benchmark suite in our experiments. We used a 16 core IA-64 RX7640, with 2 cores per socket, with each core of 1.6 GHz clock frequency and 9 MB non-shared L3 cache per core.

Table 1 describes the characteristics of the STAMP benchmarks when run on an unmodified TL2 implementation with 16 threads. It reports the number of atomic sections in each of these benchmarks, the % of aborts to total transactions started, the major data structures used by the application and the dominant atomic sections where most aborts occur. We instrumented TL2 to obtain the number of aborts for each static instance of atomic section in the application. We find that most of the aborts are accounted for by few of the atomic sections in each of the benchmarks. We report the major data structures accessed in the atomic sections in column 4 and the atomic sections where most of the aborts occur in terms of filename and line number in column 5 of Table 1. Note that STAMP benchmarks use a set of common data structures like list, queue, rb-tree, map and hash table whose implementation is provided along with the benchmark suite itself.

4.2 Performance of SCLA-STM

Table-2 gives the number of candidate data items selected for compile time lock assignment by SCLA-STM scheme for each benchmark.

Benchmark	**SCLA-STM candidates found**
Kmeans	4
Vacation	4
Genome	0
Intruder	2
Labyrinth	1
Ssca2	6
yada	3
bayes	3

Table – 2 SCLA-STM candidates

We also measured the compile time overhead due to SCLA-STM by comparing the compile time of each benchmark with SCLA-STM enabled and without SCLA enabled. We found that compile time overheads due to SCLA-STM phase were within the range of 0.9% to 2.3%.

We find that our SCLA-STM scheme had runtime performance impact on only 4 of the 8 benchmarks namely kmeans, intruder, yada and vacation. *We report performance results only for these 4 benchmarks, in which SCLA-STM has a performance impact. We measured negligible performance differences (< ±1%) on applying SCLA-STM to the other 4 benchmarks (we discuss the reasons for this later in the section).* We report both the % improvement in execution time and the % reduction in aborts, for 2,4,8 and 16 threads in Table-3 for the 4 STAMP benchmarks on which SCLA-STM had a performance impact (of > 1%).

In the benchmark 'Kmeans', SCLA-STM improves performance by 11% to 37%. Kmeans has 3 atomic sections. All the shared data referenced in these atomic sections are selected by SCLA-STM and are assigned compile time locks. The majority of the performance benefits come from the atomic section in file normal.c:168, where 2 arrays '*centre*' and '*centre_len*' are accessed. SCLA-STM associates a separate lock with each section of the array accessed in the atomic section. This helps to reduce the aborts and lock acquire overheads compared to the fine grained lock assignment by STM.

Benchmark	% Improvement in	2 threads	4 threads	8 threads	16 threads
kmeans	Exec. Time	11.15%	18.18%	32.21%	37.13%
	Aborts	21.28%	25.01%	25.12%	29.9%
intruder	Exec. Time	2.84%	7.83%	8.33%	10.21%
	Aborts	2.31%	2.89%	2.91%	3.12%
vacation	Exec. Time	9.23%	12.51%	14.41%	24.46%
	Aborts	37.3%	44.21%	45.2%	49.7%
yada	Exec. Time	1.68%	2.23%	2.67%	3.12%
	Aborts	1.43%	1.87%	1.62%	2.19%

Table – 3 Performance Improvements due to SCLA-STM

SCLA-STM selects the *streamQueuePtr* (stream.c) and *decodedQueuePtr* (decoder.c) for compile time lock assignment and assigns each of them a separate lock. SCLA-STM improves performance by 2.8% to 10.21%. In vacation, there is no single atomic section that contributes to most of the aborts. Instead all the three atomic sections contribute to the aborts. The main data structures are the 4 reservation tables in the manager structure, which are implemented through RB-tree, which are selected by SCLA-STM for compile time lock assignment. SCLA-STM shows performance improvement of 9.2% to 24.46%. In Yada, SCLA-STM selects the shared data items '*globalworkHeapPtr*', '*global_totalNumAdded*' and '*global_numProcess*' for compile time lock assignment. Since the atomic sections involving them are not hot, the performance improvement is minor (1.68% to 3.12%).

Though SCLA-STM scheme finds candidates in labyrinth, ssca2 and bayes for compile time lock assignment as shown in Table-2, these benchmarks have negligible performance improvements over unmodified STM. We found that the candidates selected by SCLA-STM scheme occur in atomic sections which are cold in these benchmarks. For instance, in the benchmark 'labyrinth', SCLA-STM assigns compile time lock to '*workQueuePtr*'. However the atomic section where this datum is accessed (router.c:379) is cold and hence SCLA-STM has no impact on the application performance for this benchmark. Similar cases occur in SSCA2 and bayes as well.

Note that the results in Table-3 are with compiler assigned locks placed externally to the structures protected by them. As mentioned in Section 3.8, we can use the compiler's structure layout optimization phase [25] to modify the structure layout to insert the lock in-place inside the data structure itself for compiler assigned locks. Placing the compiler assigned locks in-place in the structure reduces the data cache misses encountered in accessing the locks, compared to the case where the locks are external to the structure. Our compiler is able to perform this optimization for 3 of the candidate benchmarks intruder, vacation and yada. It could not perform this optimization for kmeans since the data structure in kmeans is not a structure but an array. We report the reduction in data cache misses when locks are co-located with the data structure, as compared with that of the external lock placement in Table-4 when these benchmarks are run with 16 threads. Data cache miss information was obtained using the performance profile tool HP-Caliper [24] by sampling the IA-64 hardware performance counters. We note that CLA can help co-locate the locks with data and hence help improve memory performance.

Benchmark	% reduction in dcache miss lat.cycles with internal locks as compared to external locks
Intruder	4.37%
Vacation	5.26%
Yada	0.72%

Table – 4 Reduction in Cache miss latency cycles with In-Place Locks

4.3 Runtime Overheads

We measured the overheads due to setting up and querying of the Compiler Lock Assignment Table. In this experiment for measuring the overhead, SCLA-STM scheme inserts the lock mapping and STM queries CLAT for the SCLA-STM candidates, but the STM does not use the compiler assigned lock, instead it uses the TM's assigned lock mapping using the hash function for shared data accesses. The overheads measured as the performance degradations over our base line (run with 16 threads), are shown in Table-5. We find that overhead ranges from 0.34% to 8.33%. While more efficient designs for CLAT are possible, our prototype maintains CLAT as a separate table for different address ranges of the application address space. Overhead due to CLAT look up can reduce the performance benefits of SCLA-STM. It is possible to reduce the CLAT lookup overhead by having a K-entry lookup cache. We are investigating this as part of our future work.

Benchmark	**Overhead**
kmeans	8.33 %
vacation	6.12 %
genome	0.00 %
intruder	3.91 %
labyrinth	0.34 %
Ssca2	0.56 %
Yada	1.47 %
bayes	0.91 %

Table – 5 Runtime Overhead Due to SCLA-STM

5. Related Work

Implementation of atomic sections using compile time lock assignment schemes has been studied in [1, 2, 3, 4, 5, 6]. Some of these approaches require user annotations [7]. Autolocker [7] takes the programs annotated with pessimistic atomic sections and a programmer controlled lock assignment, and infers a compiler controlled lock assignment that is free of deadlocks and data races. Our SCLA scheme requires no programmer annotations other than the atomic section specifications.

Emmi et al. [2] formulate the lock allocation problem as an ILP problem which minimizes the conflict cost between atomic sections and minimizes the number of locks. Hicks et al. [4] have proposed a lock inference technique for atomic sections, which first determines a set of shared memory locations in the program, then find the dependence relation among shared memory locations, and partition the shared memory locations into sets according to this dependence relation. Locks are then assigned to each memory location set. Sreedhar et al [1, 6] propose a compiler inferred lock assignment scheme which assigns minimum number of locks to critical sections by solving the Minimum Lock Assignment (MLA) problem which is formulated as an ILP Problem. They also propose a heuristic solution. Cherem et al. [3] propose a compile time lock assignment scheme which can support multi-granularity locks.

For indirect data accesses using pointers, the above schemes need to make a conservative approximation of the data items pointed to by the pointer, which is highly dependent on the alias analysis employed by the compiler. Since these schemes need to infer a lock assignment to all shared data items of the application and can not apply compile time lock assignment selectively to shared data items like our selective lock assignment approach. Further their lock assignment can degrade performance in shared data accesses with large may-alias sets. Also these compile time lock assignment schemes explicitly add the instrumentation required to acquire the locks on entry to the atomic section unlike our hybrid approach which only generates the lock mapping and lets the lock acquire be handled by the underlying STM implementation as per its optimistic concurrency algorithm.

By being a selective approach, our SCLA-STM scheme enables us to reap benefits associated with the CLA schemes while defaulting back to STM's fine grained locking for shared

data accesses which are not amenable to compiler analysis. While it is not the intent of our current work to evaluate our hybrid scheme by comparison with a full-fledged compile time lock assignment scheme, we show that a selective compile time lock assignment scheme acting in aid to STM's fine grained locking can help improve performance.

The impact of False conflicts in STM have been studied in [14, 15] and hash function improvements to reduce false conflicts have been suggested in [16]. However these schemes are unaware of the application's data access patterns and hence cannot take advantage of it unlike our approach which is based on compiler's inter-procedural analysis of the application. Riegel et al [22] propose using compiler analysis to identify and hence construct the data partitions which exhibit different characteristics so that various STM policies can be tuned independently for each partition. A partition can be tuned at runtime from being read-only to various states like transaction-local or thread-local. Based on runtime profile information on number of aborts and access frequency, they can associate different concurrency control with each partition such as shared lock, exclusive lock or multiple locks. However their scheme needs runtime profile information for setting the concurrency control at runtime.

6. Conclusion

So far, we have discussed our SCLA-STM scheme where the sole intention is aid the STM's runtime fine grained lock assignment scheme by assigning locks at compile time for certain shared data selectively using compiler's knowledge of the data access patterns inside atomic sections. We have implemented a prototype of our SCLA-STM scheme in IA-64 hp-ux C/C++ compiler using TL2 as our underlying STM implementation. We showed that our SCLA-STM scheme can reduce aborts and improve application performance from 1.67% to 37.1% for certain benchmarks.

References

1. Y. Zhang, V. Sreedhar, W. Zhu, V. Sarkar, and G. Gao. Optimized lock assignment and allocation: A method for exploiting concurrency among critical sections. TR-CAPSL-TM-065, University of Delaware, Newark, DE, 2007.
2. M. Emmi, J. S. Fischer, R. Jhala, and R. Majumdar. Lock allocation. In POPL'07: Proceedings of the 34th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 291–296, 2007.
3. S. Cherem, T. Chilimbi, and S. Gulwani. Inferring locks for atomic sections. Technical Report MSR-TR-2007-111, MSR, August 2007.
4. M. Hicks, J. Foster, and P. Pratikakis. Lock inference for atomic sections. In TRANSACT'06: Proceedings of the 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing, 2006.
5. R. L. Halpert, C. J. F. Pickett, and C. Verbrugge. Component-based lock allocation. In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, September 2007.
6. Y. Zhang, V.C. Sreedhar, W. Zhu, V. Sarkar, and G. R. Gao. 2008. Minimum Lock Assignment: A Method for Exploiting Concurrency among Critical Sections. In Languages and Compilers For Parallel Computing: 21th international Workshop, LCPC 2008, Edmonton, Canada, July 31 - August 2, 2008.
7. M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In Proceedings of the 20th Annual International Symposium on Computer Architecture, pages 289–300, May 1993.
8. N. Shavit and D. Touitou. Software transactional memory. In Proceedings of the 14th ACM Symposium on Principles of Distributed Computing, Aug 1995.
9. M. Herlihy, V. Luchangco, M. Moir, and W. N.Scherer, III. Software transactional memory for dynamic-sized data structures. In PODC '03: Proc. 22nd ACM Symposium on Principles of Distributed Computing, July 2003.
10. M.F. Spear, V.J. Marathe, W.N. Scherer III, and M.L. Scott, "Conflict Detection and Validation Strategies for Software Transactional Memory," Proc. of the 20th Int'l Symp. on Distributed Computing, Stockholm, Sweden, Sept. 2006.
11. D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In Proceedings of the 20th International Symposium on Distributed Computing (DISC), Stockholm, Sweeden, September 2006.
12. P. Felber, C. Fetzer, and T. Riegel,. 2008. Dynamic performance tuning of word-based software transactional memory. In Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (Salt Lake City, UT, USA, February 20 - 23, 2008). PPoPP '08. ACM, New York, NY, 237-246.
13. A. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation, pages 26–37, NY, USA, 2006.
14. C. Zilles and R. Rajwar. Transactional Memory and the Birthday Paradox. In Proceedings of the Nineteenth ACM Symposium on Parallel Algorithms and Architectures, June 2007.
15. C. Zilles, and R. Rajwar,. 2007. Implications of False Conflict Rate Trends for Robust Software Transactional Memory. In Proceedings of the 2007 IEEE 10th international Symposium on Workload Characterization - Volume 00 (September 27 - 29, 2007). IISWC. IEEE Computer Society, Washington, DC, 15-24.
16. R. Yoo, Y. Ni, A. Welc, B. Saha, A. Adl-Tabatabai, and H.S. Lee. 2008. Kicking the tires of software transactional memory: why the going gets tough. In Proceedings of the Symposium on Parallelism in Algorithms and Architectures (Munich, Germany, June 14 - 16, 2008). SPAA '08. ACM, NY, 265-274.
17. C. Cascaval, C.Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. 2008. Software Transactional Memory: Why Is It Only a Research Toy?. *Queue* 6, 5 (Sep. 2008), 46-58.
18. B. Steensgaard. Points-to analysis in almost linear time. In Proceedings of the ACM Symposium on the Principles of Programming Languages, St. Petersburg Beach, FL, Jan 1996.

19. M. Burke and R. Cytron. 1986. Interprocedural dependence analysis and parallelization. *SIGPLAN Not.* 21, 7 (Jul. 1986), 162-175.
20. C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In IISWC '08: Proc. IEEE International Symposium on Workload Characterization, pages 35–46, Sep 2008.
21. G. Chaitin. 2004. Register allocation and spilling via graph coloring. *SIGPLAN Not.* 39, 4 (Apr. 2004), 66-74.
22. T. Riegel C. Fetzer, and P. Felber. 2008. Automatic data partitioning in software transactional memories. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures* (Munich, Germany, June 14 - 16, 2008). SPAA '08. ACM, New York, NY, 152-159.
23. T. Harris and K. Fraser. 2003. Language support for lightweight transactions. In Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (Anaheim, California, USA, October 26 - 30, 2003). OOPSLA '03. ACM, New York, NY, 388-402.
24. R. Hundt, "HP Caliper: A framework for performance analysis tools," IEEE Concurrency, vol. 8, no. 4, pp. 64–71, 2000.
25. R. Hundt, S. Mannarswamy, and D. Chakrabarti, "Practical structure layout optimization and advice," in Proceedings of the International Symposium on Code Generation and Optimization, (Washington, DC, USA), pp. 233–244, IEEE Computer Society, 2006.
26. D. Dice and N. Shavit. 2007. Understanding Tradeoffs in Software Transactional Memory. In Proceedings of the international Symposium on Code Generation and Optimization (March 11 - 14, 2007). Washington, DC, 21-33.
27. S. Moon, X. D. Li, R. Hundt, D. R. Chakrabarti, L. A. Lozano, U. Srinivasan, and S.-M. Liu, "Syzygy - a framework for scalable cross-module ipo," in CGO '04: Proceedings of the international symposium on Code generation and optimization, (Washington, DC, USA), p. 65, IEEE Computer Society, 2004.